

Embedding Computations & Code

Session 3

Mickaël CANOUIL, Ph.D.^{1*}

Saturday, the 7th of February, 2026

Affiliations

¹Independent Contractor, Lille, France

ORCID

 Mickaël CANOUIL, Ph.D.: [0000-0002-3396-4549](https://orcid.org/0000-0002-3396-4549)

*Corresponding author: pro@mickael.canouil.dev

Table of Contents

1. Session Overview	4
1.1. Session 3: Embedding Computations & Code	5
1.2. Session Objectives	5
1.3. Learning Objectives	5
2. Computing Environments	6
2.1. Setup R, Python, and/or Julia for Quarto	7
2.1.1. R	7
2.1.2. Python (uv)	7
2.1.3. Julia	7
2.2. Engine Configuration	8
2.3. What is a Code Cell?	8
2.4. Code Cell Structure	8
2.5. Why Use Code Cells?	9
2.5.1. Reproducibility	9
2.5.2. Documentation	9
3. Execution Options	10
3.1. Setting Execution Options Globally	11
3.2. Code Cells Options	11
4. Cache and Freeze	13
4.1. Two Systems	14
4.1.1. Cache: Stores computation results	14
4.1.2. Freeze: Prevents re-execution	14
4.2. Cache: Development Speed	14
4.3. Freeze: Collaboration Control	14
4.4. Key Behavioural Differences	14
4.5. When to Use Each	15
4.5.1. Use Cache When	15
4.5.2. Use Freeze When	15
4.6. Essential CLI Commands	15
4.6.1. Cache Control	15
4.6.2. Freeze Management	15
4.7. Golden Rules	15
5. Parameters	16
5.1. What Are Parameters?	17
5.2. Defining Parameters	17
5.2.1. Python (papermill)	17
5.2.2. R (knitr)	17
5.3. Passing Parameters: Command Line	18
5.4. Passing Parameters: YAML Files	18
5.5. Practical Example	19

- 5.5.1. Setup (Jupyter/Python) 19
 - 5.5.2. Using parameters 19
 - 5.5.3. Command line usage 19
- 5.6. Best Practices 19
 - 5.6.1. ☒ Do 19
 - 5.6.2. ☒ Avoid 19
- 5.7. Workflow Integration 20
 - 5.7.1. Automated reporting 20
- 6. Hands-On Exercise: Adding Computational Power to Your Portfolio 21
 - 6.1. Exercise Overview 22
 - 6.2. Part 1: Environment Setup 22
 - 6.2.1. Python 22
 - 6.2.2. R 22
 - 6.2.3. Julia 23
 - 6.3. Part 2: Replace Static with Dynamic 23
 - 6.3.1. Python 23
 - 6.3.2. R 23
 - 6.3.3. Julia 24
 - 6.4. Part 3: Create Your First Visualization 24
 - 6.4.1. Python 24
 - 6.4.2. R 24
 - 6.4.3. Julia 24
 - 6.5. Part 4: Experiment with Code Visibility 24
 - 6.5.1. Python 24
 - 6.5.2. R 25
 - 6.5.3. Julia 25
 - 6.6. Success Criteria 25

1. Session Overview

- 1.1. Session 3: Embedding Computations & Code
- 1.2. Session Objectives
- 1.3. Learning Objectives

1.1. Session 3: Embedding Computations & Code

- Computing Environments
 - Setup R, Python, and Julia for Quarto.
 - Engine configuration and code cell structure.
 - Understanding multi-language workflows.
- Execution Options
 - Code cell options using comment + pipe syntax.
 - Control code visibility, evaluation, and output.
 - Global execution settings in YAML.
- Cache and Freeze
 - Cache for development speed vs freeze for collaboration.
 - Understanding Jupyter and Knitr caching systems.
 - CLI commands for cache and freeze management.
- Parameters
 - Creating dynamic document variations.
 - Defining parameters in Python and R.
 - Command-line parameter passing and YAML files.

1.2. Session Objectives

This session explores Quarto's computational capabilities, focusing on seamless multi-language workflows, code cell options, and integration patterns for R and Python data science stacks.

1.3. Learning Objectives

By the end of this session, participants will be able to:

- Execute R, Python, Julia code.
- Configure code cells using “comment symbol” + “pipe” (|) syntax.
- Control code visibility and execution.
- Cache computations for improved performance.

2. Computing Environments

- 2.1. Setup R, Python, and/or Julia for Quarto
 - 2.1.1. R
 - 2.1.2. Python (uv)
 - 2.1.3. Julia
- 2.2. Engine Configuration
- 2.3. What is a Code Cell?
- 2.4. Code Cell Structure
- 2.5. Why Use Code Cells?
 - 2.5.1. Reproducibility
 - 2.5.2. Documentation

2.1. Setup R, Python, and/or Julia for Quarto

2.1.1. R

```
# Install renv if not already installed
install.packages("renv")

# Initialise project
renv::init()

# Install packages
renv::install("rmarkdown")

# Save current state
renv::snapshot()
```

2.1.2. Python (uv)

```
# Install uv (if not installed)
pip install uv

# Initialise project
uv init --no-package --vcs none

# Create virtual environment
uv venv

# Activate environment
source .venv/bin/activate

# Add packages
uv add jupyter papermill
```

2.1.3. Julia

```
# Start Julia in your project directory
using Pkg

# Activate current directory as project
Pkg.activate(".")
```

```
# Add packages
Pkg.add("IJulia")

# Instantiate environment
Pkg.instantiate()
```

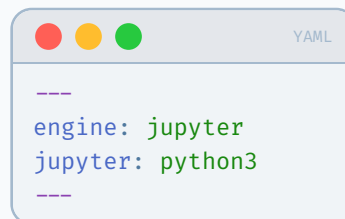
2.2. Engine Configuration

Configure engines in your document header:

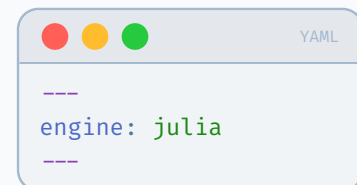
- knitr
- jupyter
- julia



```
---
engine: knitr
---
```



```
---
engine: jupyter
jupyter: python3
---
```



```
---
engine: julia
---
```



Tip

Set explicitly the engine in your document header instead of relying on auto-detection which does not work for inline code (i.e., ``{language}` ... ``).

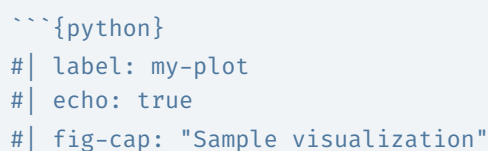
2.3. What is a Code Cell?

A code cell (also called a code chunk) is a section of executable code (defined by ``{language}``) embedded within your Quarto document.

Code cells allow you to:

- Execute programming code directly in your document.
- Display both the code and its output.
- Create reproducible analyses and reports.

2.4. Code Cell Structure



```
```{python}
#| label: my-plot
#| echo: true
#| fig-cap: "Sample visualization"
```

QMD

①

②

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Sine Wave")
plt.show()
'''
```

③

- ① Computing language (e.g., `python`, `r`, `julia`).
- ② Code cell options (e.g., `label`, `echo`, `eval`).
- ③ Code content (your actual programming code).

## 2.5. Why Use Code Cells?

### 2.5.1. Reproducibility

- Code and results stay together.
- Easy to re-run analyses.
- Version control friendly.

### 2.5.2. Documentation

- Combine narrative and code.
- Show methodology clearly.
- Create living documents.

## 3. Execution Options

- 3.1. Setting Execution Options Globally
- 3.2. Code Cells Options

## 3.1. Setting Execution Options Globally

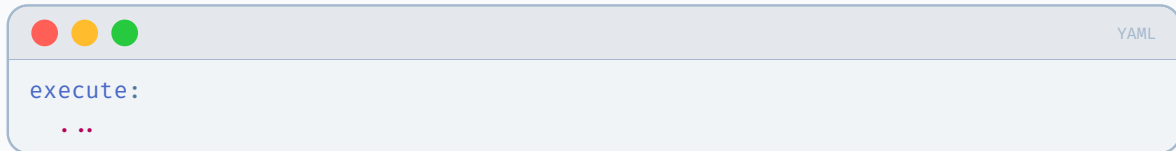


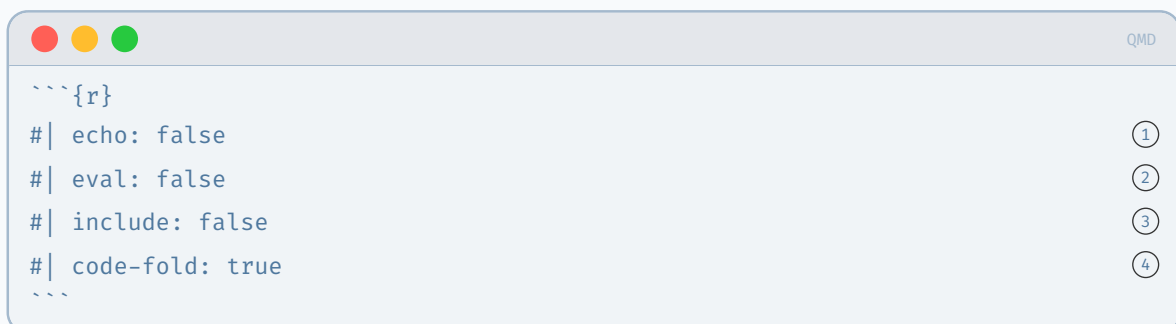
Table 3.1: `execute` options

Option	Description
<code>eval</code>	Evaluate the code chunk (if <code>false</code> , just echos the code into the output).
<code>echo</code>	Include the source code in output
<code>output</code>	Include the results of executing the code in the output ( <code>true</code> , <code>false</code> , or <code>asis</code> to indicate that the output is raw markdown and should not have any of Quarto's standard enclosing markdown).
<code>warning</code>	Include warnings in the output.
<code>error</code>	Include errors in the output (note that this implies that errors executing code will not halt processing of the document).
<code>include</code>	Catch all for preventing any output (code or results) from being included (e.g., <code>include: false</code> suppresses all output from the code block).
<code>renderings</code>	Specify rendering names for the plot or table outputs of the cell, e.g., <code>[light, dark]</code> . See <a href="#">Renderings</a> .

## 3.2. Code Cells Options

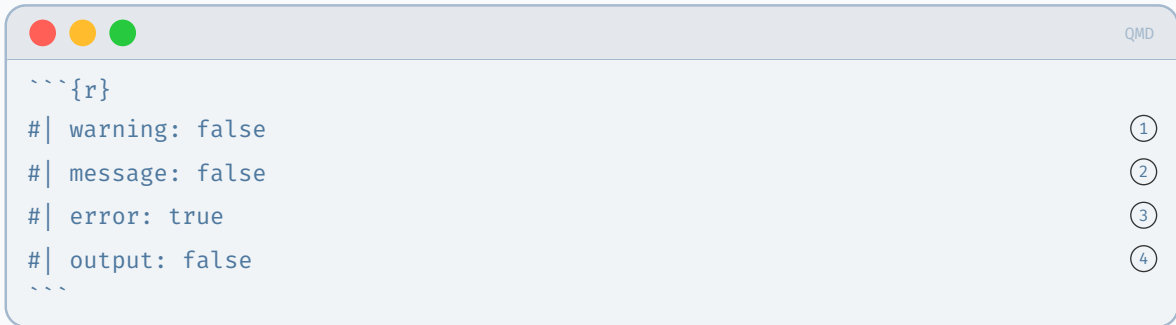
The “comment symbol” + “pipe” ( `#|` ) syntax allows to pass options to code cells.

Code Visibility Control:



- ① Hide code, show output.
- ② Show code, don't execute.
- ③ Execute code, hide everything.
- ④ Collapsible code blocks.

Output Control:



```
```{r}
#| warning: false
#| message: false
#| error: true
#| output: false
```
```

- ① Suppress warnings
- ② Suppress messages
- ③ Show errors for debugging
- ④ Hide all output

## 4. Cache and Freeze

### 4.1. Two Systems

4.1.1. Cache: Stores computation results

4.1.2. Freeze: Prevents re-execution

### 4.2. Cache: Development Speed

### 4.3. Freeze: Collaboration Control

### 4.4. Key Behavioural Differences

### 4.5. When to Use Each

4.5.1. Use Cache When

4.5.2. Use Freeze When

### 4.6. Essential CLI Commands

4.6.1. Cache Control

4.6.2. Freeze Management

### 4.7. Golden Rules

## 4.1. Two Systems

### 4.1.1. Cache: Stores computation results

- Speeds up development.
- Works during iteration.
- Engine-specific behaviour.

### 4.1.2. Freeze: Prevents re-execution

- Enables collaboration.
- Project-wide control.
- Version control friendly.

## 4.2. Cache: Development Speed

Store expensive computation results to avoid re-running during development.

```
execute:
 cache: true
```

Two Types:

- Jupyter Cache: Document-level (all-or-nothing).
- Knitr Cache: Cell-level (granular control).

## 4.3. Freeze: Collaboration Control

Prevent documents from re-executing during project renders.

Results saved in `_freeze` directory (commit to version control).

```
execute:
 freeze: auto # Re-render only when source changes
 freeze: true # Never re-render during project render
```

## 4.4. Key Behavioural Differences

| Action                     | Cache                                    | Freeze                                                 |
|----------------------------|------------------------------------------|--------------------------------------------------------|
| Individual document render | <input checked="" type="checkbox"/> Used | <input checked="" type="checkbox"/> Ignored            |
| Project render             | <input checked="" type="checkbox"/> Used | <input checked="" type="checkbox"/> Controls execution |
| Prose changes              | No effect                                | No effect                                              |
| Code changes               | Invalidates cache                        | Triggers re-render (auto mode)                         |

## 4.5. When to Use Each

### 4.5.1. Use Cache When

- Developing multi format documents.
- Iterating on analysis.
- Working with expensive computations.
- Editing prose frequently.

### 4.5.2. Use Freeze When

- Working in teams.
- Managing large projects.
- Environment setup is complex.
- Results need to be reproducible.

## 4.6. Essential CLI Commands

### 4.6.1. Cache Control

```
quarto render --cache # Force cache use
quarto render --no-cache # Disable cache
quarto render --cache-refresh # Force refresh
```

### 4.6.2. Freeze Management

```
quarto render document.qmd # Always executes (ignores freeze)
quarto render # Respects freeze settings
```

## 4.7. Golden Rules

1. Cache for Development.  
Use when iterating on individual documents.
2. Freeze for Collaboration.  
Use when working with teams or managing large projects.
3. Different Scopes.
  - Cache: Session-level performance.
  - Freeze: Project-level workflow management.
4. Version Control.  
Always commit `_freeze` directory contents.

## 5. Parameters

- 5.1. What Are Parameters?
- 5.2. Defining Parameters
  - 5.2.1. Python (`papermill`)
  - 5.2.2. R (`knitr`)
- 5.3. Passing Parameters: Command Line
- 5.4. Passing Parameters: YAML Files
- 5.5. Practical Example
  - 5.5.1. Setup (Jupyter/Python)
  - 5.5.2. Using parameters
  - 5.5.3. Command line usage
- 5.6. Best Practices
  - 5.6.1. ☒ Do
  - 5.6.2. ☐ Avoid
- 5.7. Workflow Integration
  - 5.7.1. Automated reporting

## 5.1. What Are Parameters?

Parameters allow you to create dynamic variations of the same document:

- Geographic reports → Different locations
- Time-based analysis → Different periods
- Scenario modelling → Different assumptions
- Multi-client reports → Different datasets



Tip

Think of parameters as variables that make your documents code reusable and flexible.

## 5.2. Defining Parameters

### 5.2.1. Python (papermill)

```
#| tags: [parameters]
alpha = 0.1
ratio = 0.5
country = "UK"
year = 2024
```

- Must use `#| tags: [parameters]` comment.
- Provide sensible default values.
- Parameters available in top-level environment.
- Works with `.qmd` and `.ipynb` files.

### 5.2.2. R (knitr)

```

title: "Sales Report"
params:
 region: "North"
 start_date: "2024-01-01"
 threshold: 1000

```

Access in R code:

```
print(params[["region"]])
```

## 5.3. Passing Parameters: Command Line

Use the `-P` ( `--execute-param` ) flag to override parameter values:

```
Single parameter
quarto render report.qmd --execute-param country:France

Multiple parameters
quarto render report.qmd -P year:2023 -P region:Europe

With output specification
quarto render report.qmd -P country:Germany --to typst
```

## 5.4. Passing Parameters: YAML Files

Create reusable parameter sets with YAML files:

```
country: "Spain"
year: 2023
budget: 50000
```

```
quarto render report.qmd \
 --execute-params params.yml
```

### 💡 Benefits

Version control, team sharing, complex configurations.

## 5.5. Practical Example

### 5.5.1. Setup (Jupyter/Python)

```
```python
#| tags: [parameters]
region = "All"
quarter = "Q1"
min_sales = 1000
```
```

### 5.5.2. Using parameters

```
```python
if region != "All":
    df = df[df['region'] == region]

df_filtered = df[df['sales'] >=
min_sales]
title = f"{region} Sales -
{quarter}"
```
```

### 5.5.3. Command line usage

```
quarto render sales.qmd \
 --execute-param region:North \
 --execute-param quarter:Q2
```

#### Results

Customised report for North region, Q2 data.

## 5.6. Best Practices

### 5.6.1. ☒ Do

- Provide meaningful defaults.
- Use descriptive names.
- Document parameter types.
- Validate values in code.
- Use YAML for complex configs.

### 5.6.2. ☐ Avoid

- Generic names ( `x`, `val` ).
- Parameters without defaults.
- Hardcoded values.
- Overly complex structures.

## 5.7. Workflow Integration

### 5.7.1. Automated reporting

```
Daily reports
quarto render report.qmd -P date:"$(date +%Y-%m-%d)"

Multiple outputs
for region in North South East West; do
 quarto render regional.qmd -P region:$region
done
```

💡 Perfect for

CI/CD pipelines, batch processing, scheduled reports.

## 6. Hands-On Exercise: Adding Computational Power to Your Portfolio

---

- 6.1. Exercise Overview
- 6.2. Part 1: Environment Setup
  - 6.2.1. Python
  - 6.2.2. R
  - 6.2.3. Julia
- 6.3. Part 2: Replace Static with Dynamic
  - 6.3.1. Python
  - 6.3.2. R
  - 6.3.3. Julia
- 6.4. Part 3: Create Your First Visualization
  - 6.4.1. Python
  - 6.4.2. R
  - 6.4.3. Julia
- 6.5. Part 4: Experiment with Code Visibility
  - 6.5.1. Python
  - 6.5.2. R
  - 6.5.3. Julia
- 6.6. Success Criteria

## 6.1. Exercise Overview

Objective: Transform your Session 2 portfolio document into a computational showcase by adding live code execution, demonstrating the power of embedding computations in Quarto documents using modern Python tools.

Example Code: [Exercises](#)

```
tar -xzf "03-exercises.tar.gz" -C "03-embedding-computations-code"
```

## 6.2. Part 1: Environment Setup

Choose your language and update your YAML header:

### 6.2.1. Python

```

title: "My Computational Portfolio"
author: "Your Name"
engine: jupyter
jupyter: python3
execute:
 echo: true
 warning: false

```

Install packages:

```
uv add polars plotnine # pip install polars plotnine
```

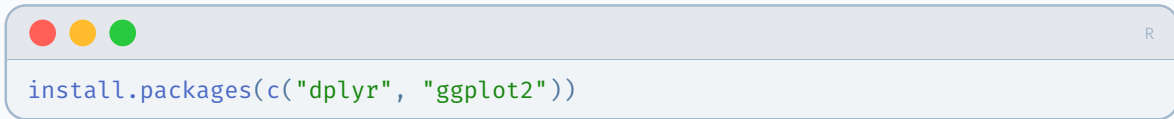
### 6.2.2. R

```

title: "My Computational Portfolio"
author: "Your Name"
engine: knitr
execute:
 echo: true
 warning: false

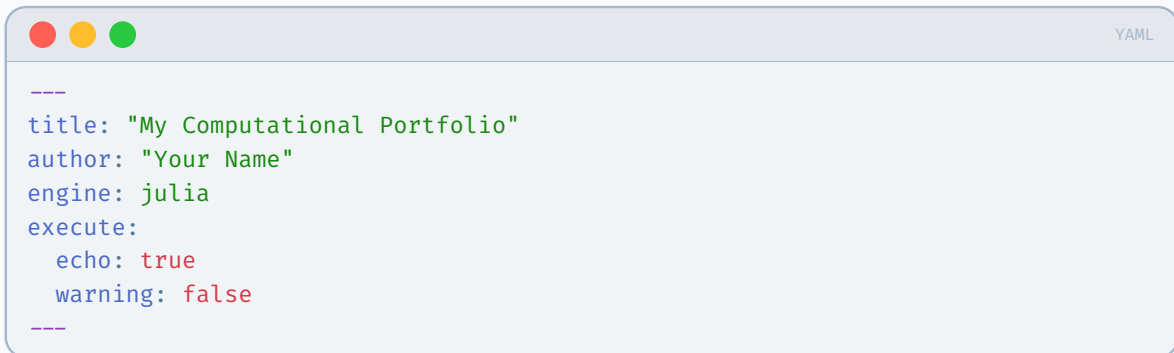
```

Install packages:



```
install.packages(c("dplyr", "ggplot2"))
```

## 6.2.3. Julia

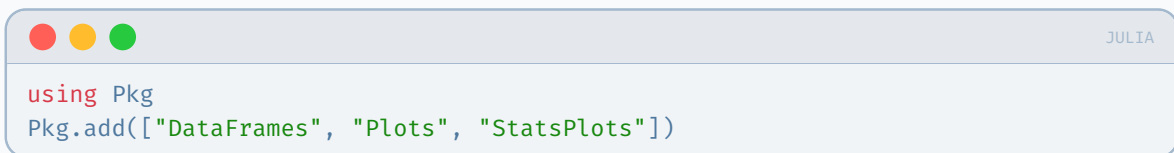


```

title: "My Computational Portfolio"
author: "Your Name"
engine: julia
execute:
 echo: true
 warning: false

```

Install packages:



```
using Pkg
Pkg.add(["DataFrames", "Plots", "StatsPlots"])
```

## 6.3. Part 2: Replace Static with Dynamic

Replace your skills table from Session 2:

### 6.3.1. Python

- Create a Polars `DataFrame` with your skills data.
- Include columns: skill name, confidence level, hours practiced.
- Use `print()` to display the data nicely.
- Calculate total practice hours with `pl.sum()`.
- Show the result using inline code ``{python} your_calculation``.

### 6.3.2. R

- Create a `data.frame` with your skills data.
- Include columns: skill name, confidence level, hours practiced.
- Use `knitr::kable()` to display the table nicely.
- Calculate total practice hours with `sum()`.
- Show the result using inline code ``{r} your_calculation``.

### 6.3.3. Julia

- Create a `DataFrame` with your skills data.
- Include columns: skill name, confidence level, hours practiced.
- Display the `DataFrame` directly.
- Calculate total practice hours with `sum()`.
- Show the result using inline code ``{julia} your_calculation``.

## 6.4. Part 3: Create Your First Visualization

Make a chart of your skills:

### 6.4.1. Python

- Use `plotnine` to create a bar chart.
- Try `geom_col()` or `geom_bar()`.
- Add proper labels with `labs()`.
- Use `theme_minimal()` for clean styling.
- Add a figure caption with `#| fig-cap: .`

### 6.4.2. R

- Use `ggplot2` to create a bar chart.
- Try `geom_col()` or `geom_bar()`.
- Add proper labels with `labs()`.
- Use `theme_minimal()` for clean styling.
- Add a figure caption with `#| fig-cap: .`

### 6.4.3. Julia

- Use `Plots.jl` to create a bar chart.
- Try `bar()` function.
- Add proper labels with `xlabel!()` and `ylabel!()`.
- Add a title with `title!()`.
- Add a figure caption with `#| fig-cap: .`

## 6.5. Part 4: Experiment with Code Visibility

Try different code cell options:

### 6.5.1. Python

- Create a cell with `#| eval: false` (shows code, doesn't run).
- Make a collapsible section with `#| code-fold: true`.
- Hide code with `#| echo: false` but show output.
- Use `#| include: false` for data setup.

## 6.5.2. R

- Create a cell with `#| eval: false` (shows code, doesn't run).
- Make a collapsible section with `#| code-fold: true`.
- Hide code with `#| echo: false` but show output.
- Use `#| include: false` for library loading.

## 6.5.3. Julia

- Create a cell with `#| eval: false` (shows code, doesn't run).
- Make a collapsible section with `#| code-fold: true`.
- Hide code with `#| echo: false` but show output.
- Use `#| include: false` for package imports.

## 6.6. Success Criteria

☒ You've successfully completed the exercise if you can:

- Transform static content into dynamic, computed content using Python, R, or Julia.
- Create and display data visualisations.
- Use inline code to insert computed values into prose.
- Experiment with different code cell visibility options.